Tampere University

Julius Ikkala

SCALABLE PARALLEL PATH TRACING

for 5G Era Virtual Reality

Faculty of Information Technology and Communication Sciences
Bachelor's thesis
April 2019

ABSTRACT

Julius Ikkala: Scalable Parallel Path Tracing
Bachelor's thesis
Tampere University
Signal Processing
April 2019

Parallelized path tracing for real-time, low-latency content over a network connection is a potential future direction for mobile gaming and other 3D graphics consumption. We have implemented a 3D renderer that parallelizes path tracing to multiple computation devices, including remote servers in addition to local resources. This thesis describes a 3D renderer that is able to dynamically adjust to uneven and changing hardware performance and can use both local resources and server resources for path tracing and noise removal. The renderer is able to keep input-to-image latency below 50 ms, gain significant benefit from server resources and keep image quality acceptable. Quality losses from compromises made for parallelization are around 1 dB to 2.5 dB. These values point to a possibility of streaming high-quality path traced content over a sufficiently fast and high bandwidth network, such as the upcoming 5G mobile network standard. Virtual reality-compatible latency can be achieved by situating computation capacity close to the user to minimize physical limitations on delay.

Keywords: 3D graphics, distributed computing, path tracing, ray tracing, 5G, virtual reality

TIIVISTELMÄ

Julius Ikkala: Skaalautuva rinnakkainen polunjäljitys
Kandidaatintyö
Tampereen yliopisto
Signaalinkäsittely
Huhtikuu 2019

Verkkoyhteyden yli suoritettu rinnakkainen polunjäljitys reaaliaikaista ja matalaviiveistä sisältöä varten on mahdollinen tulevaisuusnäkymä 3D-grafiikan kulutukselle mobiilipeleissä ja muissa käyttötarkoituksissa. Tässä työssä kuvaillaan ja toteutetaan 3D-renderöijä, joka rinnakkaistaa polunjäljityksen useammalle laskentalaitteelle, mukaanlukien sekä etänä sijaitsevat palvelimet että paikalliset laskentaresurssit. Renderöijä pystyy dynaamisesti säätymään epätasaisiin ja muuttuviin laitteistoresursseihin ja voi käyttää sekä paikallista että palvelimella sijaitsevaa laskentatehoa polunjäljitykseen ja kohinanpoistoon. Renderöijä pystyy pitämään viiveen ohjaussignaalista valmiiseen kuvaan alle 50 millisekunnissa, hyötymään merkittävästi palvelinresursseista sekä pitämään kuvanlaadun hyväksyttävänä. Laatuhäviöt tehdyistä parallelisointikompromisseista ovat noin 1 dB–2.5 dB. Nämä lukemat osoittavat mahdolliseksi korkealaatuisen polkujäljitetyn sisällön suoratoiston riittävän nopean ja laajakaistaisen verkon, kuten tulevan 5G-mobiiliverkkostandardin ylitse. Virtuaalitodellisuusyhteensopiva viive voidaan saavuttaa sijoittamalla laskentakapasiteetti lähelle käyttäjää latenssin fyysisten rajoitteiden minimoinniksi.

Avainsanat: 3D-grafiikka, hajautettu laskenta, polunjäljitys, säteenjäljitys, 5G, virtuaalitodellisuus

PREFACE

This bachelor's thesis would not have been possible without the help of Matias Koskela, Michal Babej, Pekka Jääskeläinen and Atro Lotvonen. Thanks to you and all other members of the CPC and VGA research groups for your continued help with this project.

The research for this bachelor's thesis was funded by ECSEL JU project FitOptiVis (project number 783162).

Tampere, 30th April 2019

Julius Ikkala

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

5G	Fifth generation mobile network
API	Application Programming Interface, an interface for software to interact with other software or hardware
AR	Augmented Reality, computer-generated content superimposed on real-world image
BMFR	Blockwise Multi-Order Feature Regression, a noise reduction method for path traced content
BSDF	Bidirectional Scattering Distribution Function, a function modeling surface-light interaction
CCC	Color Cell Compression, an image compression scheme
DXT1	An image compression method with fixed compression ratio
FOV	Field Of View, view of the scene from the user's perspective
GPU	Graphics Processing Unit, device for computing graphics-related tasks efficiently
HMD	Head-Mounted Display
IMT	International Mobile Telecommunications standards
ITU	International Telecommunication Union
JPEG	Joint Photographics Experts Group, an image compression scheme
OpenCL	Open Computing Language, a framework for computing on heterogeneous platforms
Path tracing	A form of ray tracing that results in more photorealistic images
PC	Personal Computer
PCIe	Peripheral Component Interconnect Express, a physical interface standard
PNG	Portable Network Graphics, an image compression scheme
PSNR	Peak Signal-to-Noise Ratio, a measure of image quality
Ray tracing	A rendering method that follows rays of lights bouncing around in a scene
SVGF	Spatiotemporal Variance-Guided Filtering, a noise reduction method for path traced content

UHD	Ultra-High-Definition, a resolution of at least 3840 by 2160 pixels
VR	Virtual Reality, a virtual environment viewed through an HMD

1 INTRODUCTION

Ray tracing has recently enjoyed a notable boost in attention, partly thanks to commercial acceleration hardware appearing. Particularly, the idea of interactive path traced *virtual reality* (VR) content is desirable. However, this still requires expensive and bulky hardware to be realistically achieved. Using ray tracing to create specific effects, like reflections and shadows, is starting to become regular in the gaming industry.

In this thesis, we focus on using path tracing for rendering the entire scene, not just some specific parts of it. To offload the performance implications of this from the consumer, compute-heavy parts of the rendering can be done on a remote server. While many commercial video game cloud streaming solutions already exist [21] [22] [27], they require the game software to exist on the server and stream the entire final image, which makes very little use of local computation capability and poses quality and latency concerns.

The first 5G ("Fifth Generation") mobile networks are set to launch in the near future. Both bandwidth and latency are greatly improved from previous generations; 5G should be able to reach 1 millisecond latency and a downlink peak data rate of 20 GBit/s according to the IMT-2020 standard [19]. These performance numbers make more complicated remote computation feasible for real-time use cases.

The main motivation for this thesis is to test and demonstrate a work-in-progress OpenCL runtime for distributed and heterogeneous computation that is being developed by our research group. Using the runtime, a method is proposed to employ both local and cloud computing capacity to scalably path trace real-time content with low latency and high perceived quality. The method is able to dynamically determine whether to use local computation for parts of the path tracing workload or not. The rendering workload is parallelized among participating devices such that latency is minimized. Denoising, image compression and a method for distributing output precision according to human vision, called foveation, are used together to achieve low bandwidth. Artifacts caused by the compression and parallelization compromises are mitigated by foveation.

In Chapter 2, theory related to the methods used is presented. Chapter 3 outlines the proposed method for parallelizing path tracing workloads over 5G or equivalent networks. Performance and quality results and analysis are listed in Chapter 4. In Chapter 5 future work is outlined. Chapter 6 discusses previous related work. The thesis is concluded in Chapter 7.

2 THEORY

The proposed method depends on technologies from several different fields. Path tracing is a 3D graphics method that can produce highly realistic images. This in turn benefits virtual reality (VR) experiences, but unfortunately is very computation-heavy and therefore requires a lot of powerful hardware to run at interactive speeds. To meet the performance limitations of modern hardware, techniques called denoising and foveation are used.

In addition to 3D graphics, parallelization and networking have a major role in this work. Parallelization techniques are used to get the most performance out of one or more computation devices, which are often able to process multiple things simultaneously. Networking is then needed to enable the usage of such computation devices that are not installed on the user's hardware directly. 5G networking is an upcoming wireless communication standard with high enough performance to allow for the low-latency, high-bandwidth communication required. Compression techniques are used to reduce bandwidth costs.

2.1 3D Graphics

3D graphics is a form of computer graphics in which a 3D virtual environment ("scene") stored in a computer's memory is projected onto a plane that forms an image. The projection is done in such a way that the image creates an illusion of the 3D shapes within the scene. When projecting a 3D scene onto a 2D image, it is necessary to pick a point of view. This view is typically called the camera, as if it was a virtual camera taking a picture of the scene. [10, p. 2, 21] The process of forming a two-dimensional image from a virtual scene via a 3D graphics method is called rendering [2, p. 11]. 3D graphics has several applications in various fields, such as entertainment, research and medicine, among others [10, p. 4].

There are several ways to create the projection of a scene. The predominant method is called perspective projection, in which objects further away are drawn smaller and behind those nearer to the camera. When projecting onto a rectangular area, this forms a four-edged pyramid shape extending towards the view direction. Due to practical reasons, this area is also limited by a near plane and far plane. Objects between those planes are included in the projected image, but any parts outside are not. This truncated pyramid shape is called the view frustum. Its width and height are usually expressed as angles. These angles are called the horizontal and vertical field of view, respectively. [10, p. 303–

310]

Real-time 3D graphics refers to those use-cases in which the image updates continuously in such a way that it is observed as movement instead of changing still pictures. The rate at which the displayed image updates is called framerate and is typically measured as frames per second. The time that passes between user input and image update related to the input is called input delay or input latency. Low input latency is a desired trait for interactive 3D graphics. [2, p. 1]

2.1.1 Virtual Reality

Virtual reality (VR) is an immersive computer-generated environment and experience in which simulated stimulus creates the illusion of a real environment to the user. Typically, this means using 3D graphics to render images, which are then viewed through special goggles that immerse the user. These goggles are *head-mounted displays* (HMD), which are able to display different images to different eyes, simulating the parallax effect. Headphones are often used in conjunction with advanced audio processing methods to produce further spatial cues to the user in virtual reality. [18, p. 1–6]

An HMD typically consists of a head tracking apparatus, two displays (one per eye), and lenses in front of the displays, to let the user's eyes focus at a comfortable distance. Sometimes, they may also contain an eye tracking device for aiding in adaptive rendering techniques and gaze-based controlling of the virtual reality software. Modern HMDs tend to be physically linked to a powerful computer with a cable. Some wireless HMDs do exist on the market, but are not yet as common as wired ones.

Due to its interactive nature, virtual reality requires real-time 3D graphics. Along with other reasons, excessive input latency, insufficient field of view and poor framerate can cause nausea similar to motion sickness, called virtual reality sickness. As such, minimal input latency is desired and field of view and framerate should be as high as possible. [1]

2.1.2 Ray Tracing

Ray tracing is an umbrella term for 3D graphics methods that create the projection and determine colors based on how simulated light rays interact with the scene. Contrary to real life however, these rays traverse the scene in reverse; they start at the viewpoint and go towards the scene. This way such rays that never end up at the camera are not simulated. One way to form an image using these rays is to find a route to a light source. That route is then followed back to the camera and the final color is evaluated based on surface interaction with light. The surface interaction behavior is modeled by the *bidirectional scattering distribution function* (BSDF). [23, p. 4–11]

Ray casting is the simplest form of ray tracing. In it, the near plane (forming the resulting

image) is divided into a grid, matching image pixels. A ray is then started from the camera at the direction of each grid cell. This ray then intersects with the nearest surface in its direction. Ray casting ends the ray there, and evaluates the resulting pixel color based on the surface alone. Therefore, surfaces do not affect each others' lighting in ray casting. [10, p. 387–391]

The method typically referred to as ray tracing is Whitted-style ray tracing, in which each intersection can create new rays originating from the intersection point. Whitted-style ray tracing has three types of secondary rays: shadow rays, which check whether a light lighting the surface is shadowed by some object; reflection rays, simulating light reflecting from a mirror-like surface; and refraction rays, which simulate refraction through a volume. With these, sharp reflections, refractions and shadows can be rendered. [31]

2.1.3 Path Tracing

Path tracing is a more comprehensive ray tracing method. It approximates the rendering equation

$$I\left(x, x'\right) = g\left(x, x'\right)\left(\epsilon\left(x, x'\right) + \int_S \rho\left(x, x', x''\right) I\left(x', x''\right) dx''\right), \tag{2.1}$$

where x, x' and x'' are points on all surfaces S, $I(a, b)$ the intensity of light transported from b to a, $g(a, b)$ the geometry term that determines whether points b and a are occluded from each other, $\epsilon(a, b)$ is the intensity of light emitted from b to a and $\rho(a, b, c)$ determines how much of the light from point c is transported to point a through point b. [14]

In path tracing, the integral in the rendering equation is numerically approximated using Monte Carlo integration. Monte Carlo integration approximates integral $\int_S f(x)dx$ with equation

$$F_N = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}, \tag{2.2}$$

where F_N converges towards $\int_S f(x)dx$ as N grows. X_i is a random variable drawn from domain S, and $p(X_i)$ the probability distribution function for X_i in S. The variance inherent to Monte Carlo estimation causes visible noise artifacts in the resulting image. The amount of error caused by the noise is relative to $\frac{1}{\sqrt{N}}$. [29, p. 39]

In path tracing, Monte Carlo integration is implemented by picking a random direction for a new ray after the current ray has intersected with the nearest surface in its direction. There are multiple different strategies for the distribution of the random directions, which aim to reduce variance [23, p. 688]. Due to being an approximation for solving the rendering equation, path tracing creates very realistic images that allow for soft shadows,

rough surface reflections and refractions and global illumination effects such as diffuse interreflection.

2.1.4 Denoising

Path tracing is very resource-intensive, and contemporary hardware is typically able to trace fewer than 10 rays per pixel, resulting in high variance and noise. There are several methods that attempt to reduce this noise as a post-processing step for the image. Auxiliary information such as surface normals, positions and colors are often used by these algorithms in order to create an intelligent approximation of how the image would look like after it has converged.

Perhaps the simplest denoising algorithm is simply blurring the image. This causes edges of surfaces to be blurred as well, which is generally an undesirable trait. That can be avoided by using a bilateral blur, which does not blur over object edges [28]. The end result will still seem low-quality and blurry, however. Using surface material information, it is possible to blur the arriving lighting alone, which creates surface-detail-preserving results at the cost of losing high frequency details in lighting [13].

One method to effectively increase the number of samples per pixel is to accumulate information from previously rendered frames [20]. This is called reprojection or temporal accumulation. However, this only works when the content has been on-screen for a frame before the current one. Whenever the camera moves, new parts of the scene enter the view and only have samples from the current frame because previous samples have not yet been accumulated for them.

Using other data than simple color information is necessary for developing better denoisers. This other data is contained in so-called "feature buffers", which carry information like surface normal vectors, positions surface points and material information.

More advanced denoising algorithms such as *Spatiotemporal Variance-Guided Filtering* (SVGF) [25] and *Blockwise Multi-Order Feature Regression* (BMFR) [15] combine temporal accumulation with image-space methods in order to reduce noise. SVGF performs this processing using wavelet-based methods. BMFR, on the other hand, uses blockwise linear regression with feature buffer data.

2.1.5 Foveation

In order to further reduce the computational performance requirements for real-time path tracing, limitations of the human visual system can be taken into account. Foveation is a method for distributing image samples such that human visual acuity traits are taken into account. It places more samples to the area of the image that the user is looking at and fewer to areas covered by peripheral vision. An eye tracking apparatus is used for deter-

mining where the user is looking at. Foveation can be used to reduce the number of path traced samples needed for acceptable image quality, resulting in higher performance. [30]

There are multiple ways to implement foveation. One rather simple method is to render the image with a regular perspective projection, but at several different resolutions. The highest resolution rendering is placed at the gaze point, and lower resolution segments are placed around it. Another, more continuous method is mapping the view to a polar space, where the gaze point is located at the origin. This way, the resolution change can be made more gradual and controllable. One method using polar mapping is visual polar mapping, which adjusts regular polar mapping such that it better follows the resolution distribution of the human eye [16].

Latency requirements for foveation imposed by human traits have been researched in relation to VR. Albert et al. determined that foveation is tolerable with a delay of 50 ms to 70 ms [3]. Because of this, our targeted maximum latency is 50 ms.

2.2 Parallelization

Mobile devices such as smartphones, handheld gaming consoles and wireless VR headsets have low-power computation hardware with relatively low performance compared to high-end desktop *personal computers* (PC). As such, they are unlikely to be suitable for real-time path tracing on their own in the near future. Furthermore, even contemporary high-end *graphics processing units* (GPU) are generally unable to generate real-time path traced content at common but very high resolutions like UHD (3840x2160 pixels). Therefore, multiple devices must be utilized concurrently to achieve real-time performance. This is called parallelization. Parallelization can occur at multiple levels; within a single computation device that is able to process multiple operations simultaneously, between computation devices installed to a single computer and even over the network.

It is possible to parallelize path tracing-related work in multiple different ways. One rather typical solution in offline uses, such as animated movie rendering, is to split the screen into smaller squares ("tiles") and then divide those tiles between computation devices. Choosing this division with the goal of increasing efficiency is called load balancing. In this case, it can be implemented by work stealing, which means that each device is given a queue of tiles to render. Once a device is out of tiles, it can 'steal' more of them from other devices' queues. Another method is to simply divide tiles beforehand, so that all communication after the initial frame start can be avoided.

2.3 Networking

Separate computers connected to each other such that they are able to communicate over the connection form a network. Networking can be used to transfer information and perform calculations on a different computer than where the results are needed. Typically, the computer that needs and fetches information from another is called the client or terminal. The computer that serves that information to the client is called a server.

Especially for low-latency use cases, there are physical limitations to the distance between networked computers. Upcoming mobile networks allow for low-latency and high bandwidth communication wirelessly, which in conjunction with strategically placed servers may power future 3D graphics rendering and consumption.

2.3.1 Cloud and Edge Computing

Cloud, *edge* and *cloud-edge computing* are emerging terms for various network setups and due to their recent nature, there are multiple different definitions. In this thesis, they are used with the following definitions.

The device that the user is consuming information from is called the terminal. This could, for example, be a mobile phone or a laptop. In the context of this thesis, the terminal is assumed to have much lower computational capabilities than whatever remote computing solution it is connected to. On the other hand, computation devices installed on the terminal will have lower latency to each other than ones used over a network. Even within the terminal, there can be multiple computation devices with different latencies. Some can be directly connected to each other with a very fast link such as in a "System on a Chip" solution, where the computation devices are on the same physical chip. At the same time, other computation devices may be connected over a slower peripheral bus, such as the PCIe standard.

Cloud computing simply means deferring computations to another computer over the internet. Typically, a large number of server computers form a cluster to be used as the "Cloud". These clusters are often called data centers due to their size and relative proximity to each other. Data centers are often quite far away from the terminal, which causes additional latency. Because information is unable to propagate faster than the speed of light, a distance of 1500 km between the terminal and the data center already causes approximately 5 ms of unavoidable latency. For interactive content, both the input and the response suffer from latency overhead. For a symmetric connection, this round-trip latency is double the one-way latency overhead.

To minimize network latency and the physical limitations on latency, it is better to situate the server closer to the terminal. This setup is called "edge computing". "Cloud-edge

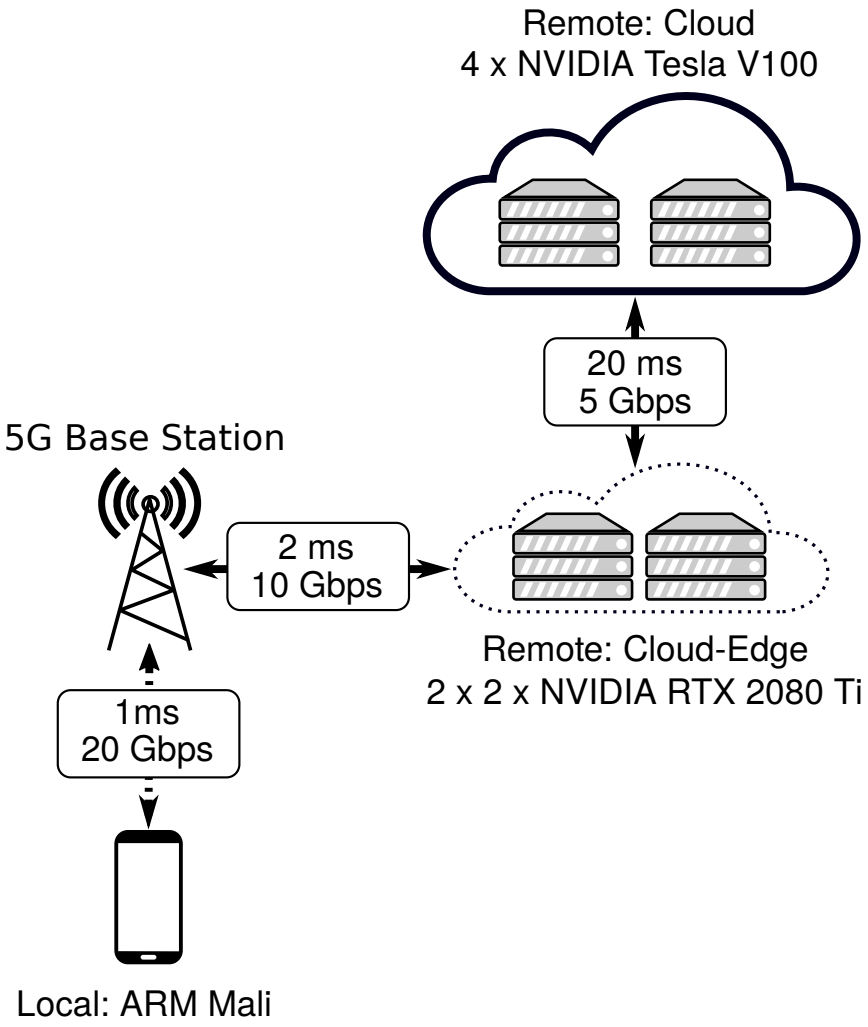


Figure 2.1. *An example network layout and device configuration for a mobile phone application using cloud-edge computation.*

computing" is the act of using both edge and cloud computation simultaneously to provide a better service that allows for low-latency actions to be performed on the edge and larger workloads on the cloud. Figure 2.1 presents an example configuration for this.

2.3.2 5G Networking

The upcoming 5G mobile network is potentially very useful for distributed parallelization of computation. This is because the requirements set to it by the *International Telecommunication Union* (ITU), called IMT-2020, define the network latency at 1 millisecond and the downlink peak data rate at 20 GBit/s [19]. These values make 5G suitable for real-time and data-intensive applications, such as path tracing.

For mobile gaming, 5G presents an opportunity for streaming VR or AR content from a cloud or edge computer [5]. To deliver graphically high-quality game content to these de-

vices, two things are required: low latency, to avoid distracting delay; and high bandwidth, to be able to deliver high-fidelity video data of the graphics. 5G should be able to provide both, given a suitable environment.

2.3.3 Compression

Oftentimes, it is necessary to reduce the amount of memory required to represent given data, which in turn saves computer memory and network bandwidth. This is done with compression. Compression methods reduce the memory usage by exploiting patterns in the data and sometimes modifying details to better suit the compression. There are two categories: lossless compression, which means that the original data can be perfectly reconstructed from the compressed representation; and the lossy compression, in which some of the information can be lost.

Compression ratio is defined as

$$R = \frac{U}{C}, \tag{2.3}$$

where R is the compression ratio, U is the uncompressed size and C is the compressed size. Lossy compression methods are able to reach higher compression ratios at the cost of accuracy.

2.3.4 Image Compression

Image compression is the field of compression algorithms specific to image data. There are several different lossless and lossy compression schemes for images, such as the lossless PNG (Portable Network Graphics) [6] and the lossy JPEG (Joint Photographics Experts Group) [12] formats.

Patterns present in image data such as large, relatively smooth areas (e.g. sky or wall) lend themselves to compression quite well thanks to their predictable nature. In addition to such exploiting patterns, lossy algorithms also make subtle modifications to the image in such a way that they are hard to spot for the human visual system. One of these modifications is to split luminance and chroma, which represent visual brightness and color hue, and store them at different precision. The eye is more sensitive to brightness details, so those are preserved at a higher precision than color data.

Texture compression is a subclass of image compression algorithms. In addition to memory and bandwidth saving, they focus on fast decompression speed and random access, which is makes them fit for use as a 3D graphics surface texture image compression algorithm [2, p. 174]. Because of this connection, GPUs often have hardware or API-level implementations of these, making decompression even faster. Current hardware texture

compression algorithms are lossy with a fixed compression ratio. This is caused by the random access requirement, which favors the ability to compute the location of each pixel in the compressed data beforehand, resulting in predictable size. This predictable size then forces the compression to be lossy, since any given data must be compressed to a predefined size.

An early form of texture compression is *color cell compression* (CCC) [7]. In it, the image is divided into 4x4 sized blocks. Then, pixels in those blocks are divided into two groups by their luminance value. This luminance can be calculated with equation

$$y = 0,30 \cdot r + 0,59 \cdot g + 0,11 \cdot b, \tag{2.4}$$

where r, g and b refer to the red, green and blue channel values for the pixel, respectively. Once divided into those two groups, the average color of the pixels in a group is calculated and assigned to them. This makes the algorithm lossy. Space is saved by then storing each pixel with just one bit, which indicates which group it belongs to.

S3 Texture Compression is a more modern approach to texture compression, and consists of several different algorithms for storing different kinds of image data. One of these is DXT1, which divides the image into 4x4 blocks, just like CCC. Instead of just two colors per block, DXT1 stores two colors and interpolates two more based on them. Each pixel in the block is then stored as two bits, which are enough to select one of the four available colors. Determining which colors are stored is up to the implementation. DXT1 has a fixed compression ratio of 6. [2, p. 174]

3 METHOD

The implementation mostly focuses on parallelizing our existing path tracing renderer and optimizing network performance by implementing buffer compression. Foveation and image denoising using BMFR [15] and SVGF [25] were implemented earlier by other members of the VGA research group.

Distribution of the computation is achieved with a special OpenCL implementation that exposes remote computation devices from a server similarly to those that are local to the device. This lets the program use those remote devices without requiring, but still allowing for special consideration for network latency and bandwidth. Figure 3.1 presents an overview of the whole pipeline. In this thesis, partitioning and load balancing is added to the renderer. Additionally, compression, transfer and merging stages are also added to the rendering pipeline.

3.1 Parallelization

Path tracing itself is a so-called 'embarrassingly parallel' problem in that pixels are unrelated to each other and can therefore be computed in parallel without synchronization issues. However, due to spatial and temporary locality benefiting GPU cache architectures, grouping several nearby rays into a 'work group' usually results in greater performance. These work groups are usually picked in squares, which are called tiles. The optimal tile size depends on the computation device.

In order to balance the path tracing workload at runtime over multiple devices, these tiles are divided between the devices. In our implementation, we decided to distribute those tiles according to predetermined boundaries. Tiles crossing the boundary are split in two along the boundary. Figure 3.2 depicts this method. Compared to work-stealing, this minimizes network communication, since only at the start and end of each frame is synchronization needed. However, it is more difficult to distribute the workload such that no device spends much time idling.

A simple load balancer is implemented for this purpose. The load balancer moves this boundary according to the time each device spent doing actual work as opposed to idling. For the benefit of the denoisers exploiting temporal accumulation, these boundaries only move one pixel per frame to avoid large slices of pixels missing information from the previous frame. Each boundary is adjusted according to neighboring load. If the load

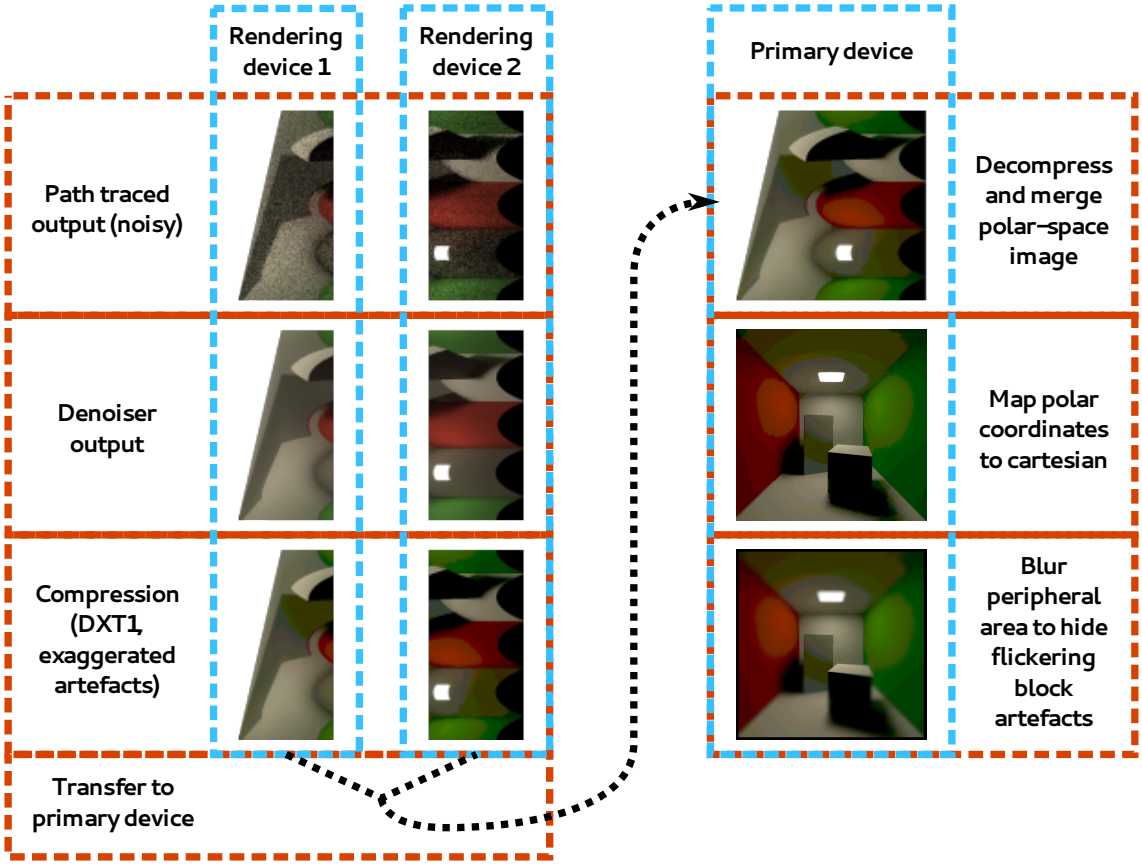


Figure 3.1. *The proposed rendering pipeline.*

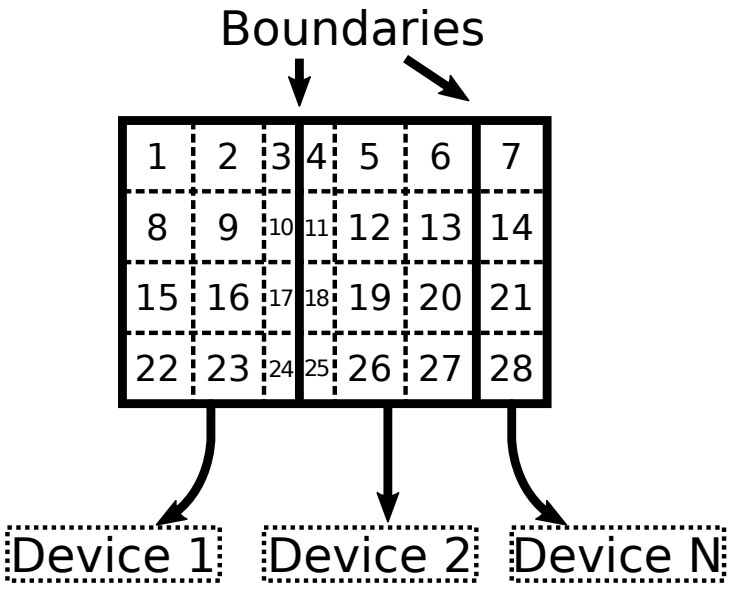


Figure 3.2. *Tile distribution to partitions along boundaries.*

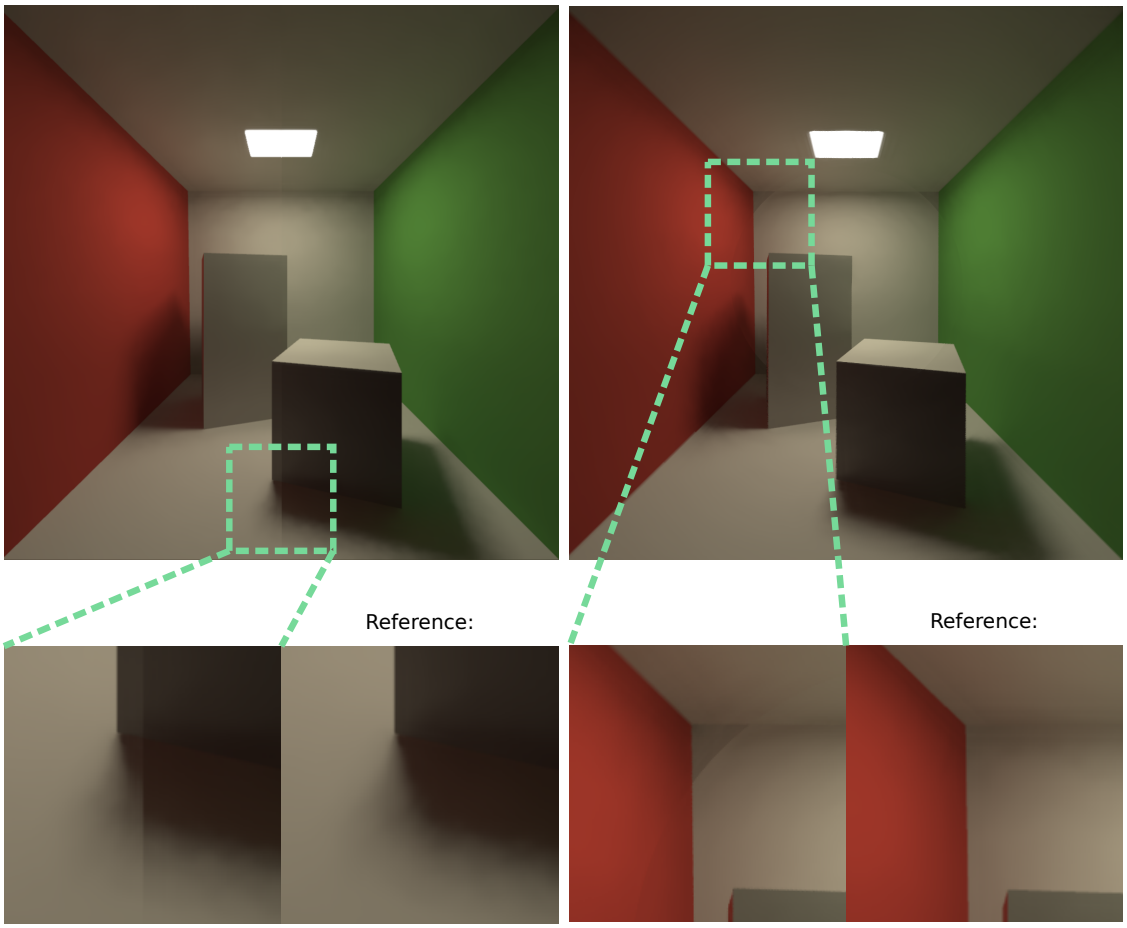


Figure 3.3. *Seam-like artifact along boundary between denoised partitions. Left: vertical boundary in cartesian coordinates, right: circular boundary in polar coordinates.*

difference between two sides of the boundary is above a given limit, the boundary is moved towards the side with higher load. This causes the slower-finishing partition to shrink, which balances the load.

Additionally, if the load balancer eventually determines that it is faster to not use a participating device for rendering at all, it is completely disabled for that purpose. This happens when its partition has completely shrunk to zero width. A separate fast path for zero-width partitions is implemented that completely disregards the device in load balancing, compression and frame synchronization from then on.

Unfortunately, neither of the two denoising algorithms available in our implementation is parallelizable as easily. For each denoised pixel, they use information from nearby pixels. Near boundary regions, splitting work to partitions means that some pixels that would be used for denoising are no longer available. This causes a visible seam-like artifact at the boundary, as seen in Figure 3.3.

One device is selected as a primary device. For performance and networking reasons, this device is the local one, e.g. the main GPU on the terminal. This device is responsible for merging partitions in a single image and executing post-processing steps. These post-processing steps are mapping the foveated image from polar coordinates to the

cartesian coordinate system supported by displays and blurring the low-resolution area of the resulting cartesian image. Once the final image has been formed in this way, it is displayed on the screen of the terminal device. Figure 3.1 contains these steps the rightmost column.

Due to compression concerns discussed in the next section, we execute the denoising stage before merging the partitions. Compressing noise-free images preserves quality better than compressing highly noisy images [26]. Because of this, it is preferrable to denoise partitions before compressing and transferring them. Furthermore, if denoising were to be done after merging, the required feature buffers would also either have to be transferred and merged or computed separately on the primary device. This would consume either more bandwidth or computation time and scaling would be worse due to this non-distributed step.

3.2 Networking

Thanks to our OpenCL implementation, very little is done in terms of actual networking code on the program side. However, for optimizing performance, it is necessary to minimize buffer transfers between different devices. In practice, the information that has to be transferred between separate computation devices per each frame consists of which partition the device should compute the tiles of and how the camera has moved. Since the demonstration program does not have moving scene geometry, such information does not need to be transferred.

The input information is mostly transferred as OpenCL "kernel" (piece of program to be run on the target computation device) arguments. Our networked OpenCL implementation causes this to happen transparently. Output information, however, is contained in an OpenCL buffer, which is simply a regular memory buffer that the device has direct access to. Once a device is ready with path tracing and denoising, its partition is compressed using a texture compression scheme. In our current implementation, DXT1 is the main compression scheme.

After the partition has been compressed, it is sent to the primary device. Once all partitions have reached the primary device, they are decompressed and merged into a single, complete image ready to be post-processed and displayed. At this point, the image is in polar coordinates, native to the visual-polar foveated rendering. Therefore, the image size is not necessarily exactly the same as the image that will be displayed. Thanks to eye-tracking information, foveated rendering can be used to lower the resolution in the peripheral area where the user is unable to discern quality degradation. Additional bandwidth can thus be saved, as the transferred image is smaller than the final cartesian-mapped image will be. In Figure 3.1, the rightmost column depicts the mapping of a smaller polar-mapped image to a larger image in cartesian coordinates.

4 EVALUATION

For the renderer, we set our target latency as 50 ms as discussed in Chapter 2. Additionally, we target ideal linear scaling over additional compute devices. In terms of quality, we aim to minimize distracting seam and compression artifacts caused by distribution compromises.

In the next section we measure quality degradation and latency for the system. After that, these results are analysed.

4.1 Results

Our testing setup consists of 3 different computers, connected with a 40 GBit/s wired connection to simulate a best-case scenario for 5G. Machine 1 contains an AMD Radeon Vega 56, and machines 2 and 3 both have two Nvidia RTX 2080 Ti GPUs each. In the testing setup, the AMD GPU is used as the local device and a varying number (1-4) of remote Nvidia GPUs are used for computation. Every computer is running Ubuntu 18.04. The scene used in these measurements is seen in Figure 4.1. This figure is rendered using one local and one remote computation device. In the measurement scenario, the foveation is done as if the user was looking at the center of the image.

Latency measurement is done by measuring the time from the program receiving input from the user to the point when the image corresponding to that input has been drawn on the screen. This measurement does therefore not consider hardware, driver or operating system level latency, which can vary significantly between display and input devices and operating systems. In our testing setup, we estimated this environment latency to be approximately (33 ± 4) ms using a high-speed camera measuring time from button press to screen color change using a trivial test program.

The test measurements are using DXT1 for compression on remote partitions, SVGF for denoising, visual-polar foveation and a resolution of 1024x1024. Each measurement is averaged over 1000 consecutive frames. In all setups the local device is the AMD device in Machine 1. All remote devices are the Nvidia GPUs from machines 2 and 3. Table 4.1 contains the results of the measurements. The first two remote devices are from machine 2, latter two from machine 3.

Quality measurements are done using machine 2. Same settings are used as in the

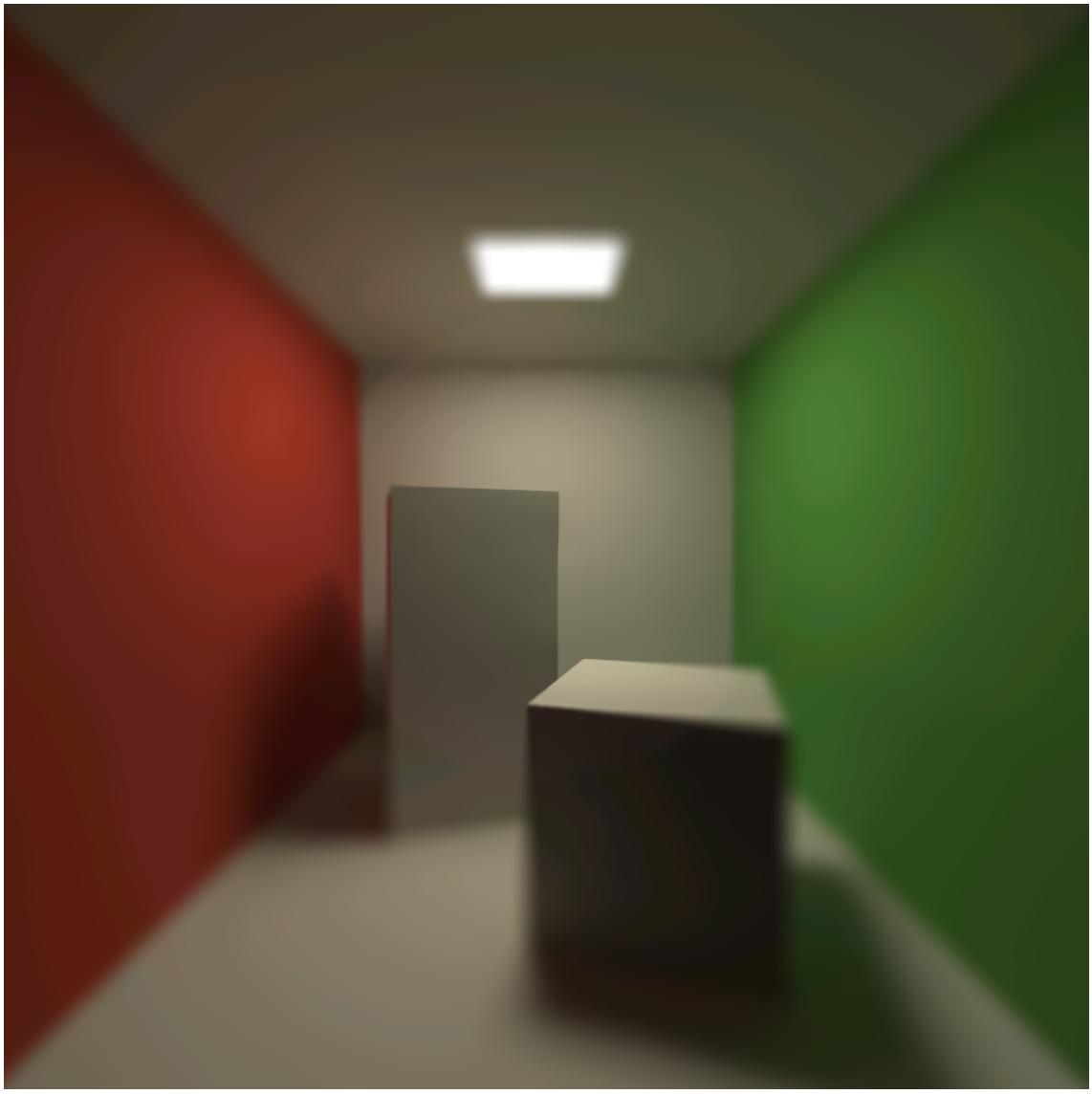
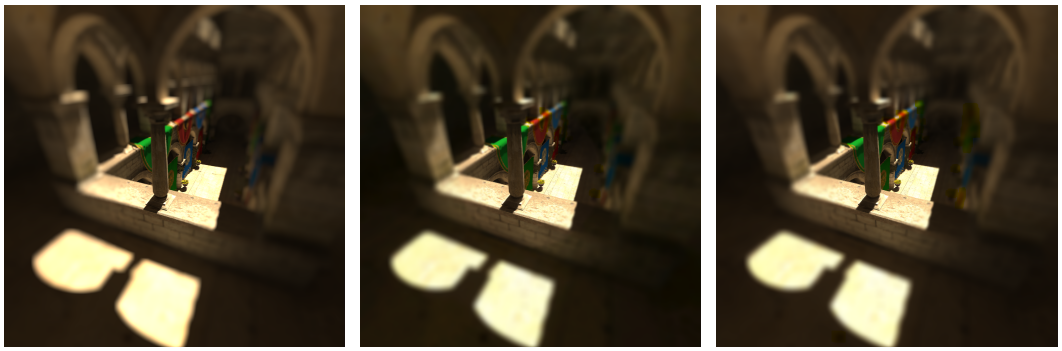


Figure 4.1. *An example output image from the renderer with Visual-Polar foveation, using DXT1 compression for the outer partition that is blurred.*

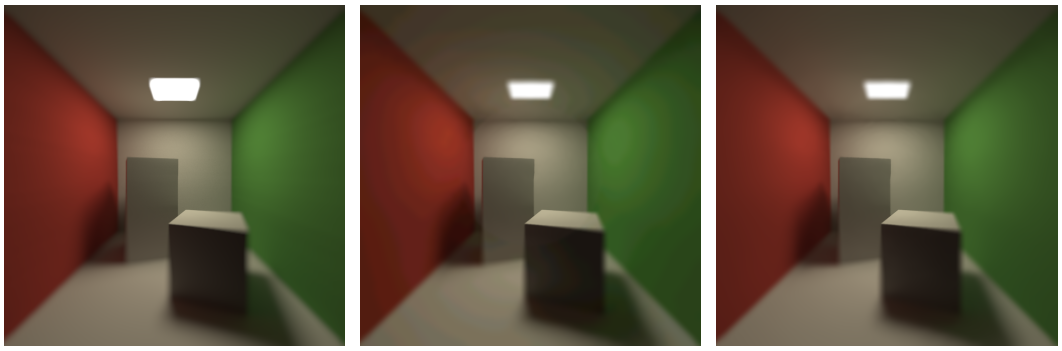
Table 4.1. *Average input-to-display latency for each setup*

Configuration	Latency (ms)
1 local, no remotes	35.0
1 local, 1 remote	25.2
1 local, 2 remote	24.7
1 local, 3 remote	23.9
1 local, 4 remote	24.0

(a) *Reference* ***(b)*** *Result, 30.37 dB* ***(c)*** *Comparison, 32.58 dB*



(d) *Reference* ***(e)*** *Result, 26.78 dB* ***(f)*** *Comparison, 27.66 dB*



(g) *Reference* ***(h)*** *Result, 29.91 dB* ***(i)*** *Comparison, 31.09 dB*

Figure 4.2. *Quality measurement of the parallelization method (PSNR).*

latency tests, but the load balancer is disabled so that the boundary location is the same in all test pictures with tiling enabled. Three different scenes are used: the classroom scene [17], Sponza scene [17] and the same Cornell box scene as shown previously. For each scene, the result image is compared to a reference image using 4096 path traced samples per pixel. The results are compared to images with no partitions or compression in Figure 4.2.

Using a setup consisting of machines 1 and 2, pipeline timing is measured. This timing is presented in Figure 4.3, with the measurement being done after load balancing has stabilized. The pipeline order is such that at the beginning of each frame, the local device first handles finished partition data from the previous frame and starts working on the next frame once the final image has been displayed. The idle time at the end of each

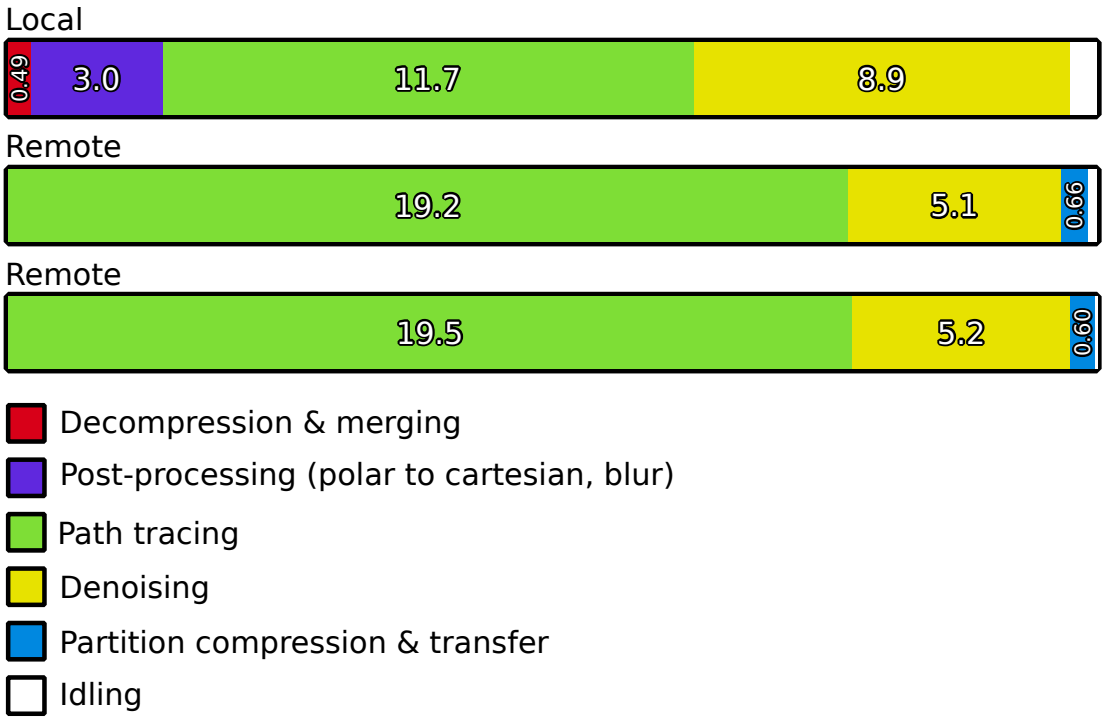


Figure 4.3. *Timing of the pipeline. All numbers are times in milliseconds taken up by the pipeline stage.*

frame is the time the devices spend waiting for each other to finish. This waiting is done so that the input state for the next frame can be sampled simultaneously and to start the new frame immediately after getting the new input data. If the load balancer was working perfectly, this idle time would not exist as all devices would finish at the same time.

4.2 Analysis

Unfortunately, at this time the remotely distributing OpenCL implementation we are developing and using is work-in-progress. While the path tracing workload is somewhat distributed, performance is lacking and scaling is far from linear. As can be seen from the results, scaling is beneficial for up to three devices.

The path tracing stage of the pipeline consists of several small kernels, which would normally take a handful of microseconds to finish. The number of kernels run is not greatly affected by the number of devices, because scaling is achieved by changing the size of the work group instead. Because our OpenCL implementation currently has notable overhead per kernel, the path tracing time is dominated by that constant overhead. For example, one of the often repeated and very short kernels takes about 10 µs to run on the Nvidia OpenCL implementation locally. Our implementation spends about 120 µs executing this kernel with the same workload and hardware when used via network connection. This overhead is mostly caused by network communication from unnecessary synchronization in the implementation.

As can be seen in Figure 4.3, the denoising stage which consists of fewer but long-running kernels has benefited from scaling much more than the path tracing stage. This is demonstrated by the ratio between them being hugely shifted towards path tracing compared to the local device.

Subpar scaling is very likely affected by accidentally duplicated work as well. For example, some parts of the denoiser code are started for the whole image on each rendering device, even though it would be enough to only cover the partition area. Fixing these problems should not be immensely difficult, but does require a lot of small modifications. Another issue is the hard limit posed by the primary device, which takes around 3.5 ms for the required post-processing steps. However, we are still able to hit our latency target of below 50 ms when remote computation resources are used.

In terms of image quality, even harsh compression in the outer region looks quite acceptable after blurring. Foveation can thus be taken advantage of for streaming content, since in addition to resolution, color depth and quality compromises can also be made without large penalty. As seen in Figure 4.2, the degradation caused by compression and partitioning is quite low in terms of PSNR, around 1 dB to 2.5 dB.

The most noticeable artifact is the circular boundary, visible in Figure 4.1 near the two upper corners at the back of the box. When denoising partitions separately, this is artifact is difficult to mitigate. Of the two denoising algorithms implemented, BMFR results in a less noticeable boundary artifact compared to SVGF.

The loss of color depth caused by DXT1 is also somewhat visible, even after the blurring. This can be seen as color banding. Additionally, if the local device is unable to render the center part fast enough, that will be streamed from the remote devices as well, which causes compression artifacts to enter the sharp area of the picture and become very visible.

5 FUTURE WORK

The main limiting factor – constant overhead of about 0.1 ms per kernel execution – should be fixed or avoided to get reasonable scaling. The main cause for this is unnecessary internal event synchronization. While it would be possible to avoid it by writing the path tracer in such a way that it simply uses fewer kernels in total, the overhead should be optimized away from our distributing OpenCL implementation, because the main purpose of this work was to stress test it with a real-world workload.

The stages that are unnecessarily run for the whole image instead of the partition should be fixed. In addition to the size of a work group, OpenCL also allows setting an offset for each work group. This should make it trivial to set the size and offset such that the work group only covers the partition.

Currently, our implementation waits for a frame to finish completely until starting to render the next frame. For better framerate, new frames could be started before previous ones have been received over the network. This way, time spent transferring frames would not be wasted waiting on the network, but on the flip side, doing this could cause higher latency. To get the same input state for all partitions, the input should be sampled right after the network transfer of the first finished partition has started.

Getting rid of frame synchronization altogether would be quite difficult and perhaps unwanted because this could cause "tearing" artifacts along partition boundaries if they updated at different rates. Additionally, there would not be a clear point for sampling the input state and adjusting partition sizes for load balancing.

Artifacts resulting from parallelization compromises can be hidden quite well using foveation and blurring of peripheral area, but are still visible in motion. One reason for the artifacts in motion is the reprojection step done by the denoisers. This could be avoided by having all rendering devices transfer rendered partitions to each other in addition to the primary device only, which would enable the denoisers to use full frame data from the previous frame.

The current load balancer implementation is quite eager to move borders. This might cause it to adapt to temporary imbalances too quickly, which negatively affects the performance of following frames. A PID controller could be used to stabilize the load balancer.

One possible way to further reduce artifacts could be to create such a denoising algorithm that boundaries between separately denoised partitions would not be as visible.

Furthermore, if a path-traced-image specific compression scheme were to be designed, the noisy frames could be denoised on the primary device, which would avoid the boundary issue altogether. Hopefully, with good enough scaling, the denoising stage can be completely skipped in the future. This can be done if the image quality and frame size are large enough that the noise is no longer visible. Getting rid of the denoising stage completely would avoid potential synchronization issues related to hiding or denoising over the boundary, since there would simply be no boundary.

Even though the blurring hides a part of the compression artifacts, some are still visible. Another texture compression method with higher color depth could be used instead to avoid this kind of color banding. Video compression schemes could also be evaluated for this purpose. They are extensively used in the game streaming industry, but unfortunately there is currently no direct access to hardware video compression and decompression from our OpenCL environment.

6 RELATED WORK

Streaming ray traced content to a mobile device from a cloud computing device has been demonstrated by Intel. In their demonstrator using Intel's MIC (Many Integrated Core) architecture, they used four server computers with dozens of CPU cores each to render the ray traced image. Two methods of distributing the rendering were presented: splitting the image to tiles, which are then rendered independently by the servers; and letting different servers render subsequent frames in an alternating fashion. They also paid special attention to transferring the image to the user device and ended up using DXT1, an image compression scheme with a fixed compression ratio, reasonably good quality and fast compression/decompression speed. [24]

EA's SEED group has implemented a similar rendering distribution system in their "Halcyon" renderer. Work is distributed between GPUs by splitting the screen into partitions, which are then combined to the primary GPU and filtered there. Their system, Virtual Multi-GPU, also allows for a remote computer's GPUs to partake in rendering. The focus of the system is on development, testing and demonstration. [4]

A ray tracing workload distribution method is presented by van Antwerpen et al. [8]. Their method partitions the image into "Uniformly Shuffled Strips" and reach very good load balancing results for dozens of computation devices. Additionally, they propose using multiple different kinds of computation devices at the same time, including devices over a network connection.

A paper by Hou et al. [9] outlined three approaches for streaming VR content with a cloud/edge-based system. They focus mostly on mitigating high latency in cloud rendering. The approaches they present are streaming field-of-view (FOV) video, streaming 360-degree video and streaming six-degrees-of-freedom content.

Several companies provide game-streaming services to consumers [21][22][27]. In these services, the game is running on the remote server owned by the service provider, and the client software is sending user inputs to the server. The server then sends the image rendered by the game, usually compressed with a video compression scheme such as H.264 [11].

Our approach uses a similar rendering distribution scheme as the "Halcyon" renderer [4], but ours allows for all kinds of OpenCL-capable devices to take part in the rendering. Our custom OpenCL platform also allows us to easily integrate remote compute devices to the program, independent of their vendor and type. Both GPUs and CPUs can be

used, along with any other OpenCL compliant device. One of the main ideas behind our renderer is to be able to dynamically adapt to changes in participating devices and the network connection quality between them.

Similar to van Antwerpen et al. method [8], load balancing is done by selecting the workload for each device before rendering the frame. In our case, the workload is not distributed in strips but as tiles, which may be split into two along a boundary. Tiles are grouped by partitions delimited by the boundaries. The partitions are transferred separately to the client device and then merged there. Filtering is done before the transfer, because the used compression scheme generally suffers less quality degradation on a less noisy image. Foveation is used both to reduce the amount of needed computation and reduce the amount of data needed to transfer from the remote compute devices to the terminal.

7 CONCLUSIONS

In this thesis, an existing path tracing renderer is extended to be able to use multiple computation devices simultaneously. In addition to locally available devices, computation devices can also be used over a network connection. In order to save bandwidth for devices used over a network connection, the DXT1 image compression method is also used. Performance and scaling is measured with a varying number of network-connected devices. Additionally, image quality is measured against the previous, non-parallelized version of the renderer.

With the current renderer, we are able to reach sub-50 ms latency for interactive path traced content, but the total latency including input and display latency is very likely to be above the target. There is still a lot of work to be done for production use. While the performance scaling of the implemented parallel path tracer is somewhat promising, further optimizations are necessary.

The implementation has several sub-optimal stages, especially in denoising and post-processing, which contribute to the sub-linear scaling. This work was created as a test case for the research group's own distributing OpenCL implementation, and does suffer a great performance hit due to the work-in-progress and unoptimized nature of the implementation.

In terms of image quality, our new renderer is usually only 1 dB to 2.5 dB worse than the non-parallelized version. The cause of these errors is the boundary artifact caused by data missing from denoising, when partitions are denoised separately; and image compression, which is lossy in this case.

Still, we believe that by realizing the ideas presented in Chapter 5, it is realistic to achieve scalable real-time, interactive path traced content for mobile and VR devices by streaming over 5G in the near future.

REFERENCES

[1] M. Abrash. *What VR could, should, and almost certainly will be within two years*. Steam Dev Days. 2014.

[2] T. Akenine-Möller, E. Haines and N. Hoffman. *Real-Time Rendering*. 3rd ed. CRC Press, 2008. ISBN: 978-1-56881-424-7.

[3] R. Albert, A. Patney, D. Luebke and J. Kim. Latency Requirements for Foveated Rendering in Virtual Reality. *ACM Trans. Appl. Percept.* 14.4 (Sept. 2017).

[4] C. Barré-Brisebois and G. Wihlidal. *Modern Graphics Abstraction & Real-Time Ray Tracing*. Syysgraph Keynote. 2018.

[5] E. Bastug, M. Bennis, M. Medard and M. Debbah. Toward Interconnected Virtual Reality: Opportunities, Challenges, and Enablers. *IEEE Communications Magazine* 55.6 (June 2017).

[6] T. Boutell. *PNG (Portable Network Graphics) Specification Version 1.0*. RFC 2083. Mar. 1997. URL: `https://tools.ietf.org/html/rfc2083`.

[7] G. Campbell, T. A. DeFanti, J. Frederiksen, S. A. Joyce and L. A. Leske. Two Bit/Pixel Full Color Encoding. *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986).

[8] E. Haines and T. Akenine-Möller. *Ray Tracing Gems*. NVIDIA, 2019. ISBN: 978-1-4842-4426-5.

[9] X. Hou, Y. Lu and S. Dey. Wireless VR/AR with Edge/Cloud Computing. *26th International Conference on Computer Communication and Networks (ICCCN)*. July 2017.

[10] J. F. Hughes, A. Van Dam, M. McGuire, D. F. Sklar, J. D. Foley, S. K. Feiner and K. Akeley. *Computer Graphics. Principles and Practice*. 3rd ed. Addison-Wesley, 2014. ISBN: 978-0-321-39952-6.

[11] International Telecommunication Union. *Advanced video coding for generic audiovisual services*. ITU-T Rec. H.264. 2017.

[12] International Organization for Standardization. *Information technology – Digital compression and coding of continuous-tone still images: Requirements and guidelines*. Standard. Feb. 1994.

[13] H. W. Jensen and N. J. Christensen. Optimizing Path Tracing using Noise Reduction Filters. *World Society for Computer Graphics*. 1995.

[14] J. T. Kajiya. The Rendering Equation. *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986).

[15] M. Koskela, K. Immonen, M. Mäkitalo, A. Foi, T. Viitanen, P. Jääskeläinen, H. Kultala and J. Takala. Blockwise Multi-Order Feature Regression for Real-Time Path Tracing Reconstruction. *ACM Transactions on Graphics* (2019). Accepted for publication.

[16] M. Koskela, A. Lotvonen, P. Jääskeläinen, M. Mäkitalo, K. Petrus and T. Viitanen. Foveated Real-Time Path Tracing in Visual-Polar Space. *Computer Graphics Forum* (2019). Submitted for publication.

[17] M. McGuire. *Computer Graphics Archive*. 2017. URL: https://casual-effects.com/data/ (visited on 04/02/2019).

[18] M. Mihelj, D. Novak and S. Beguš. *Virtual Reality Technology and Applications*. Vol. 68. Jan. 2014.

[19] *Minimum requirements related to technical performance for IMT-2020 radio interface(s)*. ITU-R M.2410-0. International Telecommunication Union, 2017.

[20] D. Nehab, P. Sander, J. Lawrence, N. Tatarchuk and J. Isidoro. Accelerating real-time shading with reverse reprojection caching. Eurographics Association, 2007.

[21] NVIDIA Corporation. *GeForce NOW*. URL: https://www.nvidia.com/en-us/geforce/products/geforce-now/ (visited on 03/25/2019).

[22] Parsec Cloud, Inc. *Parsec*. URL: https://parsecgaming.com/ (visited on 03/25/2019).

[23] M. Pharr and G. Humphreys. *Physically Based Rendering. From Theory to Implementation*. 2nd ed. Elsevier, 2010. ISBN: 978-0-12-375079-2.

[24] D. Pohl. Experimental Cloud-based Ray Tracing Using Intel® MIC Architecture for Highly Parallel Visual Processing. (2011). URL: https://software.intel.com/en-us/articles/experimental-cloud-based-ray-tracing-using-intel-mic-architecture-for-highly-parallel.

[25] C. Schied, A. Kaplanyan, C. Wyman, A. Patney, C. R. A. Chaitanya, J. Burgess, S. Liu, C. Dachsbacher, A. Lefohn and M. Salvi. Spatiotemporal Variance-guided Filtering: Real-time Reconstruction for Path-traced Global Illumination. *Proceedings of High Performance Graphics*. HPG '17. ACM, 2017. ISBN: 978-1-4503-5101-0.

[26] O. K. Al-Shaykh and R. M. Mersereau. Lossy compression of noisy images. *IEEE Transactions on Image Processing* 7.12 (1998).

[27] Sony Interactive Entertainment LLC. *PlayStation Now*. URL: https://www.playstation.com/en-us/explore/playstation-now/ (visited on 03/25/2019).

[28] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. *Sixth International Conference on Computer Vision*. Jan. 1998.

[29] E. Veach. Robust Monte Carlo Methods for Light Transport Simulation. PhD thesis. 1997.

[30] M. Weier, M. Stengel, T. Roth, P. Didyk, E. Eisemann, M. Eisemann, S. Grogorick, A. Hinkenjann, E. Kruijff, M. Magnor et al. Perception-driven Accelerated Rendering. *Computer Graphics Forum*. Vol. 36. 2. 2017.

[31] T. Whitted. An Improved Illumination Model for Shaded Display. *Commun. ACM* 23.6 (June 1980).